**DIRTY READS , PHANTOM READS**

**DIRTY READS**: Reading uncommitted modifications are call Dirty Reads. Values in the data can be changed and rows can appear or disappear in the data set before the end of the transaction, thus getting you incorrect or wrong data.

This happens at **READ UNCOMMITTED** transaction isolation level, the lowest level. Here transactions running do not issue **SHARED locks** to prevent other transactions from modifying data read by the current transaction. This also do not prevent from reading rows that have been modified but not yet committed by other transactions.

To prevent Dirty Reads, READ COMMITTED or SNAPSHOT isolation level should be used.

**PHANTOM READS:** Data getting changed in current transaction by other transactions is called Phantom Reads. New rows can be added by other transactions, so you get different number of rows by firing same query in current transaction.

In **REPEATABLE READ** isolation levels Shared locks are acquired. This prevents data modification when other transaction is reading the rows and also prevents data read when other transaction are modifying the rows. But this does not stop INSERT operation which can add records to a table getting modified or read on another transaction. This leads to PHANTOM reads.

PHANTOM reads can be prevented by using **SERIALIZABLE** isolation level, the highest level. This level acquires **RANGE locks** thus preventing READ, Modification and INSERT operation on other transaction until the first transaction gets completed.

**DIRTY READS**: Reading uncommitted modifications are call Dirty Reads. Values in the data can be changed and rows can appear or disappear in the data set before the end of the transaction, thus getting you incorrect or wrong data.

This happens at **READ UNCOMMITTED** transaction isolation level, the lowest level. Here transactions running do not issue **SHARED locks** to prevent other transactions from modifying data read by the current transaction. This also do not prevent from reading rows that have been modified but not yet committed by other transactions.

To prevent Dirty Reads, READ COMMITTED or SNAPSHOT isolation level should be used.

**PHANTOM READS:** Data getting changed in current transaction by other transactions is called Phantom Reads. New rows can be added by other transactions, so you get different number of rows by firing same query in current transaction.

In **REPEATABLE READ** isolation levels Shared locks are acquired. This prevents data modification when other transaction is reading the rows and also prevents data read when other transaction are modifying the rows. But this does not stop INSERT operation which can add records to a table getting modified or read on another transaction. This leads to PHANTOM reads.

PHANTOM reads can be prevented by using **SERIALIZABLE** isolation level, the highest level. This level acquires **RANGE locks** thus preventing READ, Modification and INSERT operation on other transaction until the first transaction gets completed.

[Spring @Transactional - isolation, propagation](http://stackoverflow.com/questions/8490852/spring-transactional-isolation-propagation)

**PROPAGATION\_REQUIRED = 0**; If DataSourceTransactionObject T1 is already started for Method M1.If for another Method M2 Transaction object is required ,no new Transaction object is created .Same object T1 is used for M2

**PROPAGATION\_MANDATORY = 2**; method must run within a transaction. If no existing transaction is in progress, an exception will be thrown

**PROPAGATION\_REQUIRES\_NEW = 3**; If DataSourceTransactionObject T1 is already started for Method M1 and it is in progress(executing method M1) .If another method M2 start executing then T1 is suspended for the duration of method M2 with new DataSourceTransactionObject T2 for M2.M2 run within its own transaction context

**PROPAGATION\_NOT\_SUPPORTED = 4**; If DataSourceTransactionObject T1 is already started for Method M1.If another method M2 is run concurrently .Then M2 should not run within transaction context. T1 is suspended till M2 is finished.

**PROPAGATION\_NEVER = 5**; None of the methods run in transaction context.

**An isolation level:** It is about how much a transaction may be impacted by the activities of other concurrent transactions.It a supports consistency leaving the data across many tables in a consistent state. It involves locking rows and/or tables in a database.

**The problem with multiple transaction**

**Scenario 1**.If T1 transaction reads data from table A1 that was written by another concurrent transaction T2.If on the way T2 is rollback,the data obtained by T1 is invalid one.E.g a=2 is original data .If T1 read a=1 that was written by T2.If T2 rollback then a=1 will be rollback to a=2 in DB.But,Now ,T1 has a=1 but in DB table it is changed to a=2.

**Scenario2**.If T1 transaction reads data from table A1.If another concurrent transaction(T2) update data on table A1.Then the data that T1 has read is different from table A1.Because T2 has updated the data on table A1.E.g if T1 read a=1 and T2 updated a=2.Then a!=b.

**Scenario 3**.If T1 transaction reads data from table A1 with certain number of rows. If another concurrent transaction(T2) inserts more rows on table A1.The number of rows read by T1 is different from rows on table A1

Scenario 1 is called **Dirty reads**

Scenario 2 is called **Nonrepeatable reads**

Scenario 3 is called **Phantom reads .**

So,isolation level is the extend to which **Scenario 1 ,Scenario 2 ,Scenario 3** can be prevented. You can obtained complete isolation level by implementing locking.That is preventing concurrent reads and writes to the same data from occurring.But it affects performance .The level of isolation depends upon application to application how much isolation is required.

**ISOLATION\_READ\_UNCOMMITTED** :Allows to read changes that haven’t yet been committed.It suffer from Scenario 1 ,Scenario 2 ,Scenario 3

**ISOLATION\_READ\_COMMITTED**:Allows reads from concurrent transactions that have been com- mitted.It may suffer from Scenario 2 ,Scenario 3 . Because other transactions may be updating the data.

**ISOLATION\_REPEATABLE\_READ**:Multiple reads of the same field will yield the same results untill it is changed by itself.It may suffer from Scenario 3.Because other transactions may be inserting the data

**ISOLATION\_SERIALIZABLE**: Scenario 1,Scenario 2,Scenario 3 never happens.It is complete isolation.It involves full locking.It affets performace because of locking.

You can test using

public class TransactionBehaviour {

// set is either using xml Or annotation

DataSourceTransactionManager manager=new DataSourceTransactionManager();

SimpleTransactionStatus status=new SimpleTransactionStatus();

;

public void beginTransaction()

{

DefaultTransactionDefinition Def = new DefaultTransactionDefinition();

// overwrite default PROPAGATION\_REQUIRED and ISOLATION\_DEFAULT

// set is either using xml Or annotation

manager.setPropagationBehavior(XX);

manager.setIsolationLevelName(XX);

status = manager.getTransaction(Def);

}

public void commitTransaction()

{

if(status.isCompleted()){

manager.commit(status);

}

}

public void rollbackTransaction()

{

if(!status.isCompleted()){

manager.rollback(status);

}

}

Main method{

beginTransaction()

M1();

If error(){

rollbackTransaction()

}

commitTransaction();

}

}

You can debug and see the result with different values for isolation and propagation.

Good question, although not a trivial one to answer.

[**Propagation**](http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/transaction/annotation/Propagation.html)

Defines how transactions relate to each other. Common options

* Required: Code will always run in a transaction. Create a new transaction or reuse one if availble.
* Requires\_new: Code will always run in a new transaction. Suspend current transaction if one exist.

[**Isolation**](http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/transaction/TransactionDefinition.html)

Defines the data contract between transactions.

* Read Uncommitted: Allows dirty reads
* Read Committed: Does not allow dirty reads
* Repeatable Read: If a row is read twice in the same transaciton, result will always be the same
* Serializable: Performs all transactions in a sequence

The different levels have different performance characteristics in a multi threaded application. I think if you understand the dirty reads concept you will be able to select a good option.

Example when a dirty read can occur

thread 1 thread 2

| |

write(x) |

| |

| read(x)

| |

rollback |

v v

value (x) is now dirty (incorrect)

So a sane default (if such can be claimed) could be Read Comitted, which only lets you read values which have already been comitted by other running transactions, in combination with an isolation level of Required. Then you can work from there if you application has other needs.

A practical example where a new transaction will always be created when entering the provideService routine and completed when leaving.

public class FooService {

private Repository repo1;

private Repository repo2;

@Transactional(propagation=Propagation.REQUIRES\_NEW)

public void provideService() {

repo1.retrieveFoo();

repo2.retrieveFoo();

}

}

Had we used Required instead the transaction [will remain open](http://stackoverflow.com/questions/2754160/question-about-spring-transaction-propagation) if the transaction was already open when entering the routine. Note also that the result of a rollback could be different as several executions could take part in the same transaction.

We can easily verify the behaviour with a test and see how results differ with propagation levels

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(locations="classpath:/fooService.xml")

public class FooServiceTests {

private @Autowired TransactionManager transactionManager;

private @Autowired FooService fooService;

@Test

public void testProvideService() {

TransactionStatus status = transactionManager.getTransaction(new DefaultTransactionDefinition());

fooService.provideService();

transactionManager.rollback(status);

// assert repository values are unchanged ...

}

With a propagation level of

* Requires new we would expect fooService.provideService() was *NOT* rolled back since it created it's own sub-transaction.
* Required we would expect everything was rolled back and backing store unchanged.

Enough explanation about each parameter is given by other answers; However you asked for a real world example,here is the one that clarifies the purpose of different **propagation** options:

Supppose you're in charge of implementing a *signup service* in which a confirmation e-mail is sent to the user.You come up with two service objects, one for *enrolling* the user and one for *sending* e-mails, which the latter is called inside the first one.For example something like this:

/\* Sign Up service \*/

@Service

@Transactional(Propagation=REQUIRED)

class SignUpService{

...

void SignUp(User user){

...

SendEmail(User);

}

}

/\* E-Mail Service \*/

@Service

@Transactional(Propagation=REQUIRES\_NEW)

class EmailService{

...

void sendMail(User user){

try{

... // Trying to send the e-mail

}catch( Exception)

}

}

You may have noticed that the second service is of propagation type **REQUIRES\_NEW** and moreover chances are it throws an exceptin (SMTP server down ,invalid e-mail or other reasons).You probably don't want the whole process to roll-back, like removing the user information from database or other things; therefor you call the second service in a separate transaction.

Back to our example, this time you are concerned about the database security, so you define your DAO classes this way:

/\* User DAO \*/

@Transactional(Propagation=MANDATORY)

class UserDAO{

// some CRUD methods

}

Meaning that whenever a DAO object, and hence a potential access to db, is created, we need to reassure that the call was made from inside one of our services, implying that a live transaction should exist; otherwise an exception occurs.Therefor the propagation is of type **MANDATORY**.

You almost never want to use Read Uncommited since it's not really ACID compliant. Read Commmited is a good default starting place. Repeatable Read is probably only needed in reporting, rollup or aggregation scenarios. Note that many DBs, postgres included don't actually support Repeatable Read, you have to use Serializable instead. Serializable is useful for things that you know have to happen completely independently of anything else; think of it like synchronized in Java. Serializable goes hand in hand with REQUIRES\_NEW propagation.

I use REQUIRES for all functions that run UPDATE or DELETE queries as well as "service" level functions. For DAO level functions that only run SELECTs, I use SUPPORTS which will participate in a TX if one is already started (i.e. being called from a service function).

Transaction Isolation and Transaction Propagation although related but are clearly two very different concepts. In both cases defaults are customized at client boundary component either by using[Declarative transaction management](http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/transaction.html#transaction-declarative) or [Programmatic transaction management](http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/transaction.html#transaction-programmatic). Details of each isolation levels and propagation attributes can be found in reference links below.

[**Transaction Isolation**](http://www.oracle.com/technetwork/issue-archive/2005/05-nov/o65asktom-082389.html)

For given two or more running transactions/connections to a database, how and when are changes made by queries in one transaction impact/visible to the queries in a different transaction. It also related to what kind of database record locking will be used to isolate changes in this transaction from other transactions and vice versa. This is typically implemented by database/resource that is participating in transaction.

[**Transaction Propagation**](http://docs.oracle.com/javaee/5/tutorial/doc/bncij.html)

In an enterprise application for any given request/processing there are many components that are involved to get the job done. Some of this components mark the boundaries (start/end) of a transaction that will be used in respective component and it's sub components. For this transactional boundary of components, Transaction Propogation specifies if respective component will or will not participate in transaction and what happens if calling component already has or does not have a transaction already created/started. This is same as Java EE Transaction Attributes. This is typically implemented by the client transaction/connection manager.

**Spring Transaction Attributes**

**What are transaction attributes?**

Spring transactions allow setting up the propagation behavior, isolation, timeout and read only settings of a transaction. Before we delve into the details, here are some points that need to be kept in mind

* Isolation level and timeout settings get applied only after the transaction starts.
* Not all transaction managers specify all values and may throw exception with some non default values

**Propagation**

PROPAGATION\_REQUIRED   
This attribute tells that the code needs to be run in a transactional context. If a transaction already exists then the code will use it otherwise a new transaction is created. This is the default and mostly widely used transaction setting.

PROPAGATION\_SUPPORTS   
If a transaction exists then the code will use it, but the code does not require a new one. As an example, consider a ticket reservation system. A query to get total seats available can be executed non-transactionally. However, if used within a transaction context it will deduct tickets already selected and reduce them from the total count, and hence may give a better picture. This attribute should be used with care especially when PROPAGATION\_REQUIRED or PROPAGATION\_REQUIRES\_NEW is used within a PROPAGATION\_SUPPORTS context.

PROPAGATION\_MANDATORY   
Participates in an existing transaction, however if no transaction context is present then it throws a TransactionRequiredException

PROPAGATION\_REQUIRES\_NEW   
Creates a new transaction and if an existing transaction is present then it is suspended. In other words a new transaction is always started. When the new transaction is complete then the original transaction resumes. This transaction type is useful when a sub activity needs to be completed irrespective of the containing transaction. The best example of this is logging. Even if a transaction roll backs you still want to preserve the log statements. Transaction suspension may not work out of the box with all transaction managers, so make sure that the transaction manager supports transaction suspension

PROPAGATION\_NOT\_SUPPORTED   
This attribute says that transaction is not supported. In other words the activity needs to be performed non-transactionally. If an existing transaction is present then it is suspended till the activity finishes.

PROPAGATION\_NEVER   
This attributes says that the code cannot be invoked within a transaction. However, unlike PROPAGATION\_NOT\_SUPPORTED, if an existing transaction is present then an exception will be thrown

PROPAGATION\_NESTED   
The code is executed within a nested transaction if existing transaction is present, if no transaction is present then a new transaction is created. Nested transaction is supported out of the box on only certain transaction managers.

**Isolation**

Isolation is a property of a transaction that determines what effect a transaction has on other concurrent transactions. To completely isolate the transaction the database may apply locks to rows or tables. Before we go through the transaction levels, let us look at some problems that occur when transaction 1 reads data that is being modified by transaction 2.

* *Dirty Reads*- Dirty reads occur when transaction 2 reads data that has been modified by transaction 1 but not committed. The problem occurs when transaction 1 rollbacks the transaction, in which case the data read by transaction 2 will be invalid.
* *Non Repeatable Reads*- Nonrepeatable reads happen when a transaction fires the same query multiple times but receives different data each time for the same query. This may happen when another transaction has modified the rows while this query is in progress.
* *Phantom Reads* - Phantom reads occur when the collection of rows returned is different when a same query is executed multiple times in a transaction. Phantom reads occur when transaction 2 adds rows to a table between the multiple queries of transaction 1.

The following isolation levels are supported by spring

ISOLATION\_DEFAULT   
Use the isolation level of the underlying database.

ISOLATION\_READ\_UNCOMMITTED   
This is the lowest level of isolation and says that a transaction is allowed to read rows that have been added but not committed by another transaction. This level allows dirty reads, phantom reads and non repeatable reads.

ISOLATION\_READ\_COMMITTED   
This level allows multiple transactions on the same data but does not allow uncommited transaction of one transaction to be read by another. This level, therefore, prevents dirty reads but allows phantom reads and nonrepeatable reads. This is the default isolation setting for most database and is supported by most databases.

ISOLATION\_REPEATABLE\_READ   
This level ensures that the data set read during a transaction remains constant even if another transaction modifies and commits changes to the data. Therefore if transaction 1 reads 4 rows of data and transaction 2 modifies and commits the fourth row and then transaction 1 reads the four rows again then it does not see the modifications made by transaction 2. (It does not see the changes made in the fourth row by the second transaction). This level prevents dirty reads and non repeatable reads but allows phantom reads.

ISOLATION\_SERIALIZABLE   
This is the highest isolation level. It prevents dirty reads, non repeatable reads and phantom reads. This level prevents the situation when transaction 1 performs a query with a certain where clause and retrieves say four rows, transaction 2 inserts a row that forms part of the same where clause and then transaction 1 reruns the query with the same where clause but still sees only four rows (does not see the row added by the second transaction)

**Read Only**

The read only attribute specifies that the transaction is only going to read data from a database. The advantage is that the database may apply certain optimization to the transaction when it is declared to be read only. Since read only attribute comes in action as soon as the transaction starts, it may be applied to only those propagation settings that start a transaction. i.e. PROPAGATION\_REQUIRED,PROPAGATION\_REQUIRES\_NEW and PROPAGATION\_NESTED.

**Timeout**

Timeout specifies the maximum time allowed for a transaction to run. This may be required since transactions that run for a very long time may unnecessarily hold locks for a long time. When a transaction reaches the timeout period, it is rolled back. Timeout needs to be specified only on propagation settings that start a new transaction

**Rollback Rules**

It is also possible to specify that transactions roll back on certain exceptions and do not rollback on other exceptions by specifying the rollback rules.